Dynamic adaptation of parallel and distributed components on Grid environments

Françoise André, Jérémy Buisson, Jean-Louis Pazat

Dynamic adaptation of parallel and distributed components on Grid environments

Françoise André[1], Jérémy Buisson[2], and Jean-Louis Pazat[2]

[1] IRISA/Universit´e de Rennes 1
[2] IRISA/INSA
Campus Universitaire de Beaulieu
Avenue du G´en´eral Leclerc
35042 RENNES Cedex - France

Abstract. Wireless communication based applications always need adaptation techniques because the bandwidth and the resources of the network often change during the execution of an application. These applications are characterized by a low bandwidth, small data sets and low-end machines. At the opposite of the spectrum, Grid applications use powerful supercomputers, high bandwidth networks and process large amount of data. However, during an application run, both the bandwidth of the network and the computing resources may also vary. In this paper, we show than despite the difference between these kinds of applications, Grid applications can benefit from adaptation techniques primarily designed for wireless applications.

1 Introduction

In wireless applications, resources are a key issue. An application cannot even work if it is not able to take into account the constraints of dynamic resources. An application have to be able to modify its behavior according to the state of the environment which is reported by some hardware or software monitors. The adaptation of the behavior may be a fairly simple operation such as the adjustment of some parameters (level of data compression for example) or can be a dramatic change in the application such as the replacement of the whole code of the application. If adaptation mechanisms are embedded into the application code, it may result in the writing of an awkward and unmanageable code. Moreover, each time a new application is designed, the whole adaptation mechanism have to be though again, written and tested.

In order to separate the adaptation aspect from the application code itself the ACEEL framework [1] which provides mechanisms for adaptation policies description and implementation will be used as a basis for our study.

In most Grid applications, performance is a key issue. If an application “works” but does not take advantage of the available resources, its overall performance will be very low in many cases. Because application execution time

are critical in these environments, applications are often scheduled to the best machines at start-up, but if new computing resources become available during run-time nothing is done to take profit of this fact. This is due to the fact that Grid application are parallel codes and the design of a parallel code is so difficult by itself that programmer would not like to add adaptation mechanism. However, if an adaptation platform is used as a complement of a Grid infrastructure such as OGSA [3] which provides tools for distributed and parallel computing, the overall performance of codes may increase dramatically.

This paper describes a way to handle the problem of adapting parallel codes to varying constraints on resources based on an adaptation framework primarily designed for wireless applications.

2 Dynamic adaptation of sequential components

Wireless applications need to be able to adapt their behavior to heterogeneous and volatile resources. A clear separation between the adaptation engine and the application code makes programs easier to maintain and allows one to easily change or improve the adaptation policy. In this section we describe the ACEEL framework [1] that provides generic mechanisms for the adaptation process and for the definition of the adaptation rules. This framework has been developed in our laboratory as a successor of the MolèNE project [9].

ACEEL has been designed as a generic framework for adaptable components. It separates the adaptive aspect from the functional part of the component, as shown on figure 1.

Each ACEEL component is separated in two different levels: the base level and the meta level.

The base level contains a set of implementations called *behaviors* among which only one is active at a time and processes the incoming requests. Within this level, the *context* holds the component's state in order to make easier the change of behavior. The context is also used as the only visible interface of the component in order to be able to use implementations with different interfaces.

Each behavior have specific resource needs, so a policy has also to be provided as a set of event-based rules: each rule is a condition on the state of the environment and is associated with a reaction which might be either the activation of another *behavior* or the adjustment of some parameters of the active one. The policy is used at the meta level by a generic meta-object called the *adapter* which decides which of the available *behaviors* should be used according to the environment changes.

A monitoring engine is used by the platform to merge the observations of different resources of the environment. When a change in the characteristics or

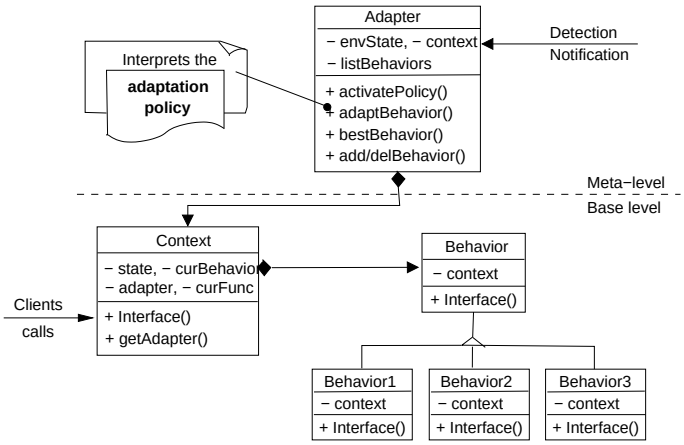

Fig. 1. Architecture of an ACEEL component

availability of a resource happens, the monitoring engine notifies the *adapter* of the components which depend on this resource according to their *adaptation policy*.

3 Parallelism and components

One of the main issues of today's programming techniques including parallel programming is to build reusable software. Current approaches rely on object oriented programming [6] and many developments now focus on component technologies [10] such as CORBA CCM [7] or Enterprise Java Beans [5]. Within these programming techniques, one of the main issues is the so called "separation of concerns" paradigm. Entities implementing distinct functionalities should be located in different modules, objects, services or components.

We think that the adaptation of a parallel program should be considered as a service for a component provided by a component server such as it is the case for transactions. The first step of this approach is to find or build a parallel component middleware/framework/architecture because the well known standard component architectures such as EJB or CORBA do not take into account the fact that an object may be implemented as a parallel code. The worst case is the EJB architecture which considers explicitly that beans cannot be multithreaded. The CORBA architecture allows to encapsulate parallel codes in CORBA objects but this leads to performance degradations: the inability of the platform to take into account the existence of several concurrent units in the component forces the platform to serialize the communications between parallel components.

Some projects, such as PARDIS [4], PaCO++ [2] and GridCCM [8], have focused on getting better performances out of a CORBA architecture. Those projects are aimed at efficiently encapsulating SPMD code into high performance CORBA objects or components. They consider a parallel object as a set of identical sequential objects. When a parallel object has to process a request, each object executes the part of the computation related to the data elements it is in charge of. Parallelism comes from the distribution of the parameters of the request. In order to rise the efficiency, an enhanced request protocol has been defined: the server allows the clients to see its internal structure and distribution. This allows the clients to send directly the data to the right sequential objects: data do not transit via a single master object anymore. This multi-port communication mode allows to efficiently use the aggregated bandwidth of the network which could not be used if only one centralized communication port was used.

These projects show that it is possible to include parallel codes into a component architecture without losing performance. In the remainder of this paper we will focus on the problem of using a generic adaptation mechanism to control and adapt parallel programs. Some practical component aspects will not be detailed but the whole framework is designed in order to be included in an existing or new component platform.

4 Modifying an adaptation platform for the Grid

The challenge is to use an adaptation framework devoted to wireless applications for parallel Grid components. The main difference is that the applications we consider are both parallel and distributed whereas wireless applications are mainly restricted to the client-server paradigm. The Grid applications considered here are built as an assembly of software components, each component being a set of communicating processes running on a cluster of workstations or on a parallel computer. We work here at the component level and we try to adapt one parallel component to environment changes; we do not consider yet the global adaptation of the whole application.

In our model, adaptation is a service that the platform gives to the components it hosts. Figure 2 shows the overall architecture of the platform hosting a parallel adaptable component.

4.1 Platform objects

The platform mainly provides two kinds of objects: the *decider* and the *coordinators*. The *decider* is the object that makes the decisions (the initiative of the adaptation and the choice of the reaction). It bases its decisions on the reports

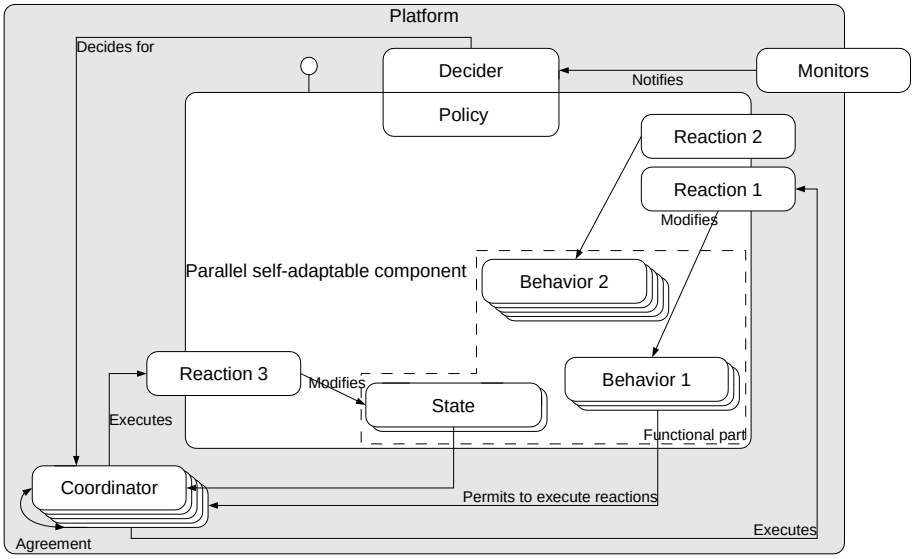

Fig. 2. Overall architecture of the adaptation framework

given by the *monitors* that track any change in the state of the environment. There is no major difference between a *decider* for wireless applications and for Grid applications, except that the set of decision rules may be more complex in the case of Grid applications.

The *coordinators* execute the directives given by the *decider*: they serve as intermediaries between the code of the component and the platform. Whereas there is no need for coordinators in the wireless case, they are of major (importance) for parallel codes. Their role is to synchronize the adaptation mechanism with the functional code and to coordinate the execution of the reactions.

4.2 Components parts

In return for the adaptation service it provides, the platform expects from the component to conform to a given structure. A component is separated in two parts : a functional part and a meta-level part.

At the functional level, a parallel adaptable component should provide to its clients services specified by the *interface* such as an ordinary component. In addition, an adaptable component should include a set of *behaviors* allowing the component to be well suited for most environment changes. Each *behavior* contains a complete implementation of the component. Contrarily to wireless applications, each *behavior* might be either a sequential or a parallel implemen-

tation. At any time, one and only one *behavior* is active, the one that processes the incoming requests. A *behavior* is a stateless object making it easier to keep the *state* in a separate place of the component. This is useful to switch easily from one *behavior* to another.

At the meta-level of the component, *reaction steps* should be provided by the programmer as the means given to the platform to modify the component: to react to a change in the environment, the component can execute on demand a sequence of *reaction steps*. Our model does not give any special semantic to these reaction *steps* except that it provides a way to modify a *behavior* or switch between *behaviors*.

The component must provide the platform with an *adaptation policy*, which is the component-specific counterpart of the *decider*. The purpose of the *adaptation policy* is to define when the adaptation mechanism should be triggered and what should be the associated reaction. It is mostly a set of event-based rules. Each rule associates a reaction to a specific event. Events are conditions on the state of the environment. For example, an *adaptation policy* can include the rule: “if the number of nodes is increased, spawn new processes and redistribute arrays”.

The behavior can be safely modified at anytime while the component is inactive, whereas it is not the case while the component is processing a request. It must be suspended in such a state that it remains consistent after the execution of the reaction. Since this consistency completely depends on the component and on its implementations, the developer has to explicitly specify those states. The developer should provide the *adaptation points* which indicate states at which the execution of reaction *steps* preserves the consistency of the component. *Reaction steps* must guaranty that those states of the component remains consistent. Adaptation points define states through which the processes of a *behavior* run. For a given component, several *behaviors* may run through a common state. In such a case, the component can resume its execution with any of these *behaviors*. The existence of those common states defines the semantic correctness of *behavior* replacement.

In the case of a parallel program, the previous definition of *Adaptation points* can be used as *local Adaptation points* to each process. They are thus not sufficient to specify global states at which the *behavior* can be modified. This is why the developer has to explicitly specify *global adaptation points* through a correspondence relationship between the *local adaptation points* of each process of the *behavior*.

4.3 Coordination of the reactions

Because *behaviors* can be parallel and *state* distributed, the reactions cannot be executed by a single central object. Each process of the active *behavior* and of the state must participate to the execution of the reactions. If the reactions are executed independently by each process, the consistency of the component cannot be guaranteed. The component can also lock itself if processes are not coordinated, trying to reach unreachable states that do not correspond to any *global adaptation point*. This can be for example the case of an SPMD code like the one shown on figure 3. Say that when they receive a request to adapt the com-

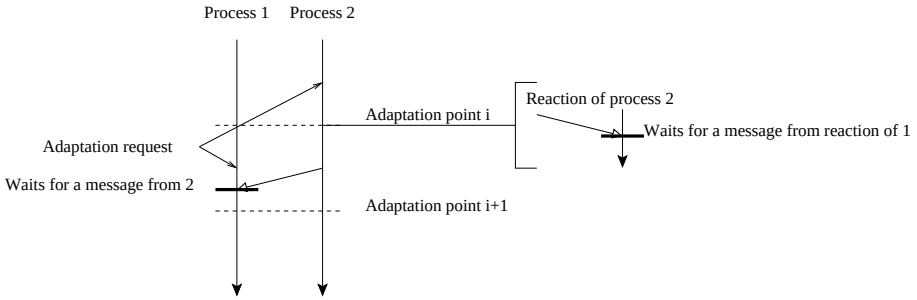

Fig. 3. Case where a component can dead-lock itself if the processes are not coordinated

ponent, one of the processes have already executed the adaptation point `i` and the second have not. Without a coordination mechanism, the first process never reach another adaptation point because the second is waiting at the adaptation point `i`, executing the reaction. The component is thus dead-locked. This is why *coordinators* have been introduced in the model. Their purpose is to coordinate the action of every processes for the execution of the reactions.

Coordinators are the intermediaries between the adaptation mechanism, the functional part of the component and the reaction. They are totally passive objects: their activity comes from both the meta-level and the functional part of the component. They can be seen as a three part object: one part is directly plugged in each process of the functional part and implements *adaptation points*; the second part is more or less a singleton that is connected to the decision maker of the component, that is to say the *decider*; the third drives the execution of the reactions.

4.4 Sketch of an adaptation

When a *monitor* reports some interesting change in the environment, for example the presence of new available nodes, it notifies the *decider*, which in turn broadcasts an adaptation initiative to the *coordinators*. If one suppose that the active *behavior* is a SPMD behavior, the *coordinators* choose the *adaptation points* before the next operation to be executed. At the end of the current operation, the *coordinators* execute the reaction: they spawn a new process on the new node, then redistribute the data. The functional code can then resume its execution and benefit from the presence of the new node.

5 Conclusion

A first experiment based on a generic SPMD code shows that the overcost of the adaptation is very low and that better performance can thus be gained through adaptation. Our application is a generic vector iteration ; vectors are distributed with a block scheme ; communications use MPI. The *adaptation policy* is to use as many nodes as the *monitor* reports, spawning new processes when nodes are inserted in the system. We placed an *adaptation point* between each iteration. In order to evaluate the gain obtained by the adaptation, we increased the number of nodes from 4 to 6 while the test application was running. The figure 4 shows the elapsed time at the end of each iteration. The execution of the reaction oc-

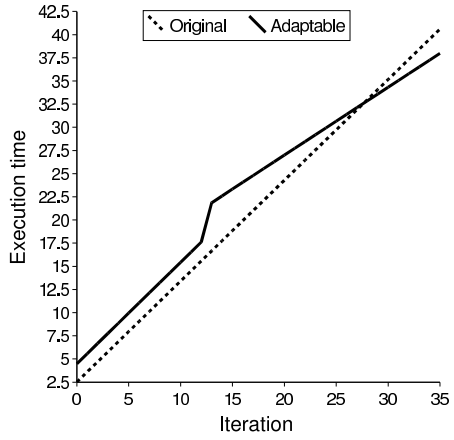

Fig. 4. Execution time of an adaptable application

curred between iterations 12 and 13 ; this appears as a break on the curve. This

figure shows that the adaptable version of the application needs some iterations after the reaction has been executed to become effectively better than the original one. This is due to the cost of the data redistribution and is not due to the platform itself. If the adaptation takes place early during the execution of the program there is a dramatic gain of performance.

This work is a first step toward the integration of a dynamic adaptation framework into a component based platform for Grid applications. In this paper, we have shown that it was possible to reuse the ideas developed for the adaptation of wireless application in a different application world. Our current prototype uses the ACEEL framework but is not yet included in a component infrastructure.

To our knowledge this is one a the first attempt to separate the adaptation mechanism from the functional code in the context of Grid applications. We think that it makes it far easier to build efficient applications for not only wireless or Grid environments but any environment with unknown and/or dynamic characteristics.

In our future works, we are planning to define more formally the properties that the component is required to satisfy in order to be able to adapt itself. This includes the properties of global states in which the adaptation is allowed to occur and the relation between fault tolerance and adaptation techniques. The constraints on behavior replacement should also be studied.

References

1. Djalel Chefrour and Françoise Andr´e. D´eveloppement d'applications en environnements mobiles à l'aide du modèle de composant adaptatif aceel. In *Langages et Modèles à Objets LMO'03. Actes publiés dans la Revue STI, série L'objet, volume 9*, Vannes, France, February 2003.
2. Alexandre Denis, Christian P´erez, and Thierry Priol. Portable parallel CORBA objects: an approach to combine parallel and distributed programming for grid computing. In *Proc. of the 7th Intl. Euro-Par'01 Conference (EuroPar'01)*, pages 835–844, Manchester, UK, August 2001. Springer.
3. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Global Grid Forum*, June 2002.
4. Katarzyna Keahey and Dennis Gannon. PARDIS: A parallel approach to CORBA. In *HPDC*, pages 31–39, 1997.
5. B. McLaughlin. *Java Enterprise Applications*, volume 1. O'Reilly, 2002.
6. B. Meyer. *Object-Oriented Software Construction, Second Edition*. Prentice Hall, 1997.
7. OMG. Corba components, June 2002. Document formal/02-06-65.
8. C. P´erez, T. Priol, and A. Ribes. A parallel corba component model. Rapport de recherche 4552, INRIA, September 2002.
9. Maria-Teresa Segarra. *Une plate-forme à composants adaptables pour la gestion des environnements sans fil*. PhD thesis, IRISA/IFSIC, 2000.
10. C. Szyperski. *Component software: beyond object oriented programming*. Addison Wesley, 1998.